INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Pragmatic Reconstruction of LambdaProlog

Catherine BELLEANNÉE, Pascal BRISSET et Olivier RIDOUX

N° 2390

Octobre 1994

PROGRAMME 2

ISSN 0249-6399

INRIA
RENNES

A Pragmatic Reconstruction of LambdaProlog

Catherine BELLEANNÉE*, Pascal BRISSET** et Olivier RIDOUX*

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Lande

Rapport de recherche n° 2390 — Octobre 1994 — 24 pages

Abstract: λProlog is a logic programming language in which hereditary Harrop formulas generalise Horn formulas, and simply typed λ-terms generalise Prolog terms. One may wonder whether these extensions are simultaneously required, and if it exists useful subsets of λProlog, at least for pedagogical purposes. We answer this question by exhibiting a network of necessity links between the new features of λProlog. A handy programming heuristic can be proposed after these necessity links for defining relations by induction on types.

Key-words: Logic programming, λ-calculus, quantifications, types, λProlog, induction

(Résumé : tsvp)

*{belleannee,ridoux}@irisa.fr
**ENAC, 7 av. Édouard Belin, BP 4005, 31055 Toulouse Cedex, pbrisset@eis.enac.dgac.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Une reconstruction pragmatique de LambdaProlog

Résumé : λProlog est un langage de programmation logique dont les clauses sont des formules héréditaires de Harrop, qui généralisent les formules de Horn, et dont le domaine de calcul est celui des λ-termes simplement typés, qui généralisent les termes de Prolog. On peut se demander si toutes ces extensions sont nécessaires simultanément et si des langages intermédiaires intéressants ne pourraient pas être définis, au moins dans un but pédagogique. Nous répondons à cette question en montrant que des liens de nécessité conduisent à adopter toutes les extensions à partir du moment où le langage de termes de Prolog est étendu à celui des λ-termes simplement typés modulo la β-équivalence. De cette reconstruction découle une heuristique de programmation par induction sur les types qui est un guide commode pour utiliser λProlog.

Mots-clé : Programmation logique, λ-calcul, quantifications, types, λProlog, induction

1 Introduction

Logic programming is a programming paradigm in which programs are logical formulas, and executing them amounts to search for a proof. The most famous practical incarnation of logic programming is Prolog, which is based on Horn formulas [25].

The formalism of Horn programs is computationally complete [1, 37], but one has often tried to augment it to gain more flexibility and expressivity. One of these attempts is λProlog [30]. It has the quality, rare among extensions to Horn formulas, to preserve a formal connection to logic [32, 31]. Indeed, a kind of goal directed proofs (proofs that can be used as an operational semantics) is complete for the formulas of λProlog.

The following equation sketchs the definition of λProlog:

$$\lambda Prolog \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms + simple\ types + =_{\alpha\beta\eta} + \forall_{\mathcal{G}} + \Rightarrow_{\mathcal{G}}$$

The components of this formula will be completely defined in the sequel. For now, it is enough to know that λ-*terms*, *simple types*, and $=_{\alpha\beta\eta}$ deal with the computing domain, while $\forall_{\mathcal{G}}$, and $\Rightarrow_{\mathcal{G}}$ deal with the structure of programs. We use notation $X_{\mathcal{Y}}$ to designate occurrences of connective X in syntactic units of type $\mathcal{Y}$ ($\mathcal{D}$ for definite clauses, and $\mathcal{G}$ for goals). For instance, Horn clause programs offer $\forall_{\mathcal{D}}$, $\wedge_{\mathcal{D}}$ implicitly, and $\Rightarrow_{\mathcal{D}}$, $\wedge_{\mathcal{G}}$ explicitly.

The first steps of a newcomer to λProlog are difficult because the relationships between all the components of λProlog are not clear, and then it is not clear how to use them. We aim at giving an explanation of the relationships via a pragmatic reconstruction of λProlog. An expected by-product is a rationale for a class of λProlog programming.

To ease the way of a beginner, it is tempting to define fragments of λProlog by merely dropping some of its components. For instance,

$$Typed\ Prolog \stackrel{\text{def}}{=} Prolog + simple\ types$$

defines a strongly typed variant of Prolog as proposed by Lakshman and Reddy [24].

$$CLP(\lambda_{\rightarrow}) \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms + simple\ types + =_{\alpha\beta}$$

defines an instance of the scheme CLP [9, 22] for the domain of the simply typed λ-terms endowed with the equivalence relation $=_{\alpha\beta}$.

$$Harrop\ Prolog \stackrel{\text{def}}{=} Prolog + \forall_{\mathcal{G}} + \Rightarrow_{\mathcal{G}}$$

defines an extension of Prolog in which connectives $\forall_{\mathcal{G}}$ and $\Rightarrow_{\mathcal{G}}$ are allowed.

$$\alpha\beta Prolog \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms + simple\ types + =_{\alpha\beta} + \forall_{\mathcal{G}} + \Rightarrow_{\mathcal{G}}$$

defines the extension to Prolog that differs from λProlog only in that η-equivalence is not considered[1].

The reconstruction we propose allows us to determine for every fragment whether it makes sense or not.

A way to help λProlog beginners is to show them heuristics and idioms for building λProlog programs. The pragmatic reconstruction we propose focuses on the notion of *essentially universal quantification*, which is also the basis of a programming heuristic.

We briefly present in section 2 the syntax and semantics of λProlog (we assume a basic knowledge of Prolog). We then argue against frequent misconceptions about the new components of λProlog (section 3). The next section (section 4) contains a necessity-driven reconstruction of λProlog. We then show how the reasoning that led to incorporate the various components of λProlog may be used to decide when to use them in programs defined by induction on a data-structure (section 5).

Every program sample will use the concrete syntax of λProlog (or of Prolog in the rare cases of Prolog samples).

[1] This definition is motivated by the fact that pure λ-calculus does not generally consider η-equivalence. Moreover, the problem of unifying λ-terms modulo $\alpha\beta$-equivalence (without η) is as well-defined as the same problem modulo $\alpha\beta\eta$-equivalence (with η).

2 λProlog

Miller and Nadathur proposed in 1986 a generalisation of the terms and formulas of Prolog which still has interesting logical and computational properties [30, 32, 31]. It encompasses Prolog as in[2]

```
type append (list T) -> (list T) -> (list T) -> o .
append [] L L .                                          % append( [], L, L ) .
append [A|L1] L2 [A|L3] :- append L1 L2 L3 .
                                  % append( [A|L1], L2, [A|L3] ) :- append( L1, L2, L3 ) .
```

but also makes possible programs with a really new structure

```
typing (abs E) (arrow A B) :- pi x\( typing x A => typing (E x) B ) .
```

in which one finds function variables like E and universal variables like x. The declarative semantics of this program is $[\forall x.typing(x,A) \Rightarrow typing(E(x),B)] \Rightarrow typing(abs(E),arrow(A,B))$. Its intentional semantics is given in section 5.2.2.

2.1 The principles of λProlog

2.1.1 The types

The new term language is the language of the simply typed λ-terms.

Simple types are generated by the following grammar:

$$\begin{array}{lll} \mathrm{T} & ::= \mathcal{U} \quad | \quad (\mathcal{K}_i\ \mathrm{T}^i) & \\ \mathrm{T} & ::= (\mathrm{T} \text{ -> } \mathrm{T}) & \text{(i)} \end{array}$$

where the $\mathcal{U}$ and $\mathcal{K}_i$ are identifiers of type variables and type constructors with arity i, respectively. Rule (i) generates function types. A type *(A->B)* can be interpreted as the type of functions whose domain is A and co-domain is B. We assume $\mathcal{K}_0$ contains at least the constant *'o'* for propositional types. We also assume the arrow, ->, associates to the right. This makes many brackets useless: for instance, *o->o->o* denotes the same type as *(o->(o->o))* does.

In λProlog, type constructors are declared using directive *kind*: for instance,

```
kind o type .                                            % o ∈ K0
kind list type -> type .                                 % list ∈ K1
```

The declaration of *list* shows it is a type constructor that must be applied to some type to actually produce a type. These declarations are standard in a concrete λProlog system.

2.1.2 The terms

Simply typed λ-terms are generated by the following grammar:

$$\begin{array}{lll} \Lambda_t & ::= \mathcal{C}_t \quad | \quad \mathcal{V}_t & t \in \mathrm{T} \\ \Lambda_{t'\text{->}t} & ::= \mathcal{V}_{t'} \setminus \Lambda_t & t,\ t' \in \mathrm{T} \quad \text{(ii)} \\ \Lambda_t & ::= (\Lambda_{t'\text{->}t} \quad \Lambda_{t'}) & t,\ t' \in \mathrm{T} \quad \text{(iii)} \end{array}$$

where the $\mathcal{C}_t$ and $\mathcal{V}_t$ are respectively identifiers of constants and identifiers of variables whose type is t. Attributes in terminal and non-terminal symbols are used to constrain types and to ensure the well-typing of generated terms. Rule (ii) generates *abstractions*, and rule (iii) generates *applications*. An abstraction can be interpreted as a function, and an application can be interpreted as the result of applying some function to some actual parameter. In this concrete syntax, abstraction is written with an infix '\' instead of the classical prefix 'λ': for instance, the identity function is written $x\backslash x$ instead of $\lambda x.x$. We assume application associates to the left, which makes some brackets useless: for instance, *(append A B C)* denotes the same term as *(((append A) B) C)* does.

In λProlog, constants and their types are declared using directive *type*: for instance,

[2] Comments give the encompassed Prolog program.

type [] (list T) . % $\forall T.\ [] \in \mathcal{C}_{(list\ T)}$
type '.' T -> (list T) -> (list T) . % $\forall T.\ '.' \in \mathcal{C}_{T\text{->}(list\ T)\text{->}(list\ T)}$
type append (list T) -> (list T) -> (list T) -> o . % $\forall T.\ append \in \mathcal{C}_{(list\ T)\text{->}(list\ T)\text{->}(list\ T)\text{->}o}$

The type of *[]* shows it is a non-functional constant. The type of *'.'* shows it is a function that takes two arguments. These two constants allows us to build *homogeneous* polymorphic lists: polymorphic lists all elements of which have the same type. Finally, the result type of *append*, *'o'*, shows it is a predicate constant.

2.1.3 A taste of λ-calculus

Abstraction leads to the notions of *heading*, *head*, and *body*. In term $a\backslash b\backslash c\backslash(b\ a\ c)$, the heading is $a\backslash b\backslash c\backslash$, the head is b, and the body is $(b\ a\ c)$. One says a term binds the variables of its heading.

One also distinguishes between *free* and *bound* occurrences of variables. In the term $(x\ y\ z\backslash\underline{x\backslash(x\ y\ z)})$, y has only free occurrences, the only occurrence of z is bound, and x has both a free occurrence and a bound one (the first and the second, respectively). In the underlined subterm, z and y have only free occurrences, and the only occurrence of x is bound. More generally, an occurrence of some variable is bound in a given term if it is a subterm of an abstraction that binds the variable and is a subterm of the given term. An occurrence of some variable is free if it is not bound. One calls *free variables* of a given term the variables that have a free occurrence in it, *bound variables* those that have a bound occurrence in the term. One writes $\mathcal{FV}(t)$ the *free variables* of t, and $\mathcal{BV}(t)$ the *bound variables*. A term without any free variable is called a *closed* term or *combinator*. One writes $[x\leftarrow y]$ the operation of replacing all free occurrences of x by y, and $E[x\leftarrow y]$ the application of this operation to term E.

The domain of simply typed λ-terms is endowed with an equivalence relation which is defined as the smallest congruence based on the following axioms:

α — $x\backslash E =_\alpha y\backslash E[x\leftarrow y]$, if y has no occurrence in E.

This axiom formalises the renaming of bound variables. For instance, $x\backslash(f\ x) =_\alpha y\backslash(f\ y)$ but $x\backslash(g\ x\ y) \neq_\alpha y\backslash(g\ y\ y)$.

β — $(x\backslash E\ F) =_\beta E[x\leftarrow F]$, if $\mathcal{FV}(F) \cap \mathcal{BV}(E) = \emptyset$.

This axiom formalises the evaluation of an application by substituting an actual parameter, F, to a formal parameter, x. For instance, $(x\backslash(f\ x)\ 72) =_\beta (f\ 72)$ but $(x\backslash y\backslash x\ y) \neq_\beta y\backslash y$ because $y \in \mathcal{FV}(y) \cap \mathcal{BV}(x\backslash y\backslash x)$. However, $(x\backslash y\backslash x\ y) =_\alpha (x\backslash w\backslash x\ y) =_\beta w\backslash y$. In fact, it is always possible to apply axiom β to a term like $(x\backslash E\ F)$ if one applies axiom α for renaming the bound variables of E.

η — $x\backslash(E\ x) =_\eta E$, if $x \notin \mathcal{FV}(E)$.

This axiom formalises the principle of *functional extensionality*: "Different functions yield different results". According to this principle, we have $[\forall x.\ (E\ x)=(F\ x)] \Rightarrow E=F$, which is not provable using axioms α and β alone. For instance, $x\backslash(f\ x) =_\eta f$, but $x\backslash((g\ x)\ x) \neq_\eta (g\ x)$. Note that axiom α can never help applying axiom η.

Axioms α and β are always present in the λ-calculus, but axiom η is optional: it gives nothing as far as the computational properties of the λ-calculus are concerned. However, we will see it is crucial for the equality theory.

2.1.4 The formulas

Clauses and goals are generated by the following grammar:

$\mathcal{P}$ ::= $\mathcal{D}$. | $\mathcal{D}$. $\mathcal{P}$ (iv)
$\mathcal{D}$ ::= $\mathcal{A}$ | $\mathcal{A}$:- $\mathcal{G}$ | *pi* $\mathcal{V}_t$ \ $\mathcal{D}$ (v)
$\mathcal{G}$ ::= $\mathcal{A}$ | $\mathcal{G}$, $\mathcal{G}$ | $\mathcal{G}$; $\mathcal{G}$ | $\mathcal{D}$ => $\mathcal{G}$ | *pi* $\mathcal{V}_t$ \ $\mathcal{G}$ | *sigma* $\mathcal{V}_t$ \ $\mathcal{G}$ (vi)
$\mathcal{A}$ ::= Λ_o (vii)

Rules (iv), (v), (vi) and (vii) generate *programs*, *clauses*, *goals* and *atomic formulas* (also called *atoms*), respectively. The novelty of λProlog is in the goal language: explicit quantifications (universal and existential, written *pi* and *sigma*) and implications (written =>) may occur in goals. Formulas generated this way are called *hereditary Harrop formulas*.

In rule (v), terminals ':-' and *pi* are the concrete writings for connective $\Rightarrow_{\mathcal{D}}$, which is already used in Prolog, and connective $\forall_{\mathcal{D}}$, implicit in Prolog. In λProlog, connective $\forall_{\mathcal{D}}$ needs only be explicit in case clauses are nested (see an example in section 2.3.4).

In rule (vi), terminals ',', ';', =>, *pi*, and *sigma* are the concrete writings for connectives $\wedge_{\mathcal{G}}$, $\vee_{\mathcal{G}}$, shared with Prolog, and connectives $\Rightarrow_{\mathcal{G}}$, $\forall_{\mathcal{G}}$, $\exists_{\mathcal{G}}$, proper to λProlog. In fact, connectives $\vee_{\mathcal{G}}$ and $\exists_{\mathcal{G}}$ can be defined at the second order using the other connectives. We mention them only because they help in making programs more readable.

Like in Prolog, a concrete program is a sequence of input units, all terminated by a full-stop, '.'. In Prolog as in λProlog, input units are clauses, and every variable that is free in an input unit is considered as universally quantified at the clause level. We call these variables *logical variables* or *unknowns*.

2.1.5 A taste of proof-theory

The semantics of λProlog is usually given in proof-theoretic terms [32, 31], as opposed to the model-theoretic semantics used for Prolog [25]. The main result is that a class of goal-directed proofs, called *uniform proofs*, is complete with respect to intuitionistic provability for hereditary Harrop formulas. In other words, every hereditary Harrop formula that is a theorem in intuitionistic logic has a uniform proof. In still other words, restricting proofs to be uniform eliminates proofs, but does not eliminate any theorem among hereditary Harrop formulas.

The operational semantics of the new connectives is as follows:

sigma — To prove a goal *(sigma v\G)*, prove goal *G[v←V]*, where *V* is a new logical variable which has the type of *v*.

pi — To prove a goal *(pi v\G)*, prove goal *G[v←c]*, where *c* is a new constant which has the type of *v*, taking care that *c* does not occur in the binding values of older logical variables.

=> — To prove a goal *D=>G*, add clause *D* to the program and prove goal *G*. Clause *D* is kept in the program during the proof of *G*. It is suppressed from it as soon as the proof of *G* is over.

It corresponds to the following deduction rules of the intuitionistic sequent calculus[3].

$$\frac{P \vdash G[x \leftarrow t]}{P \vdash \exists x.G}\ \exists_{\mathcal{G}} \qquad t \text{ is an arbitrary term.}$$

$$\frac{P \vdash G[x \leftarrow c]}{P \vdash \forall x.G}\ \forall_{\mathcal{G}} \qquad c \text{ occurs free neither in } P \text{ nor in } G.$$

$$\frac{P, D \vdash G}{P \vdash D \Rightarrow G}\ \Rightarrow_{\mathcal{G}}$$

All these rules are right introduction rules; their connective of interest is in the right part of the conclusion sequent.

2.2 Scoping constructs

The word "scope" sums-up the new constructs of λProlog:

1. abstraction limits the scope of variables in terms,
2. quantifications limit the scope of variables in formulas,
3. the deduction rules for universal quantification and implication limit the scope of constants and clauses, respectively, in the proof process.

We will show in sections 4 and 5 that programming in λProlog often amounts to make the three scoping levels interact.

[3] A sequent $P \vdash G$ reads "goal G is a consequence of program P". A rule $\frac{Sequent^*}{Sequent}$ reads "conclusion *Sequent* is true if all premises $Sequent^*$ are true".

2.3 An introductory example

We illustrate the novelties of λProlog by elaborating on the classical *grand_father* program.

2.3.1 A genealogical data-base in Prolog

We assume a Prolog data-base representing the father-child and mother-child relationships by the logical relations *father* and *mother*.

```
father( adam, cain ) .      father( adam, abel ) .      ...
mother( eve, cain ) .       mother( eve, abel ) .       ...
```

2.3.2 Introduction of λProlog syntax

To transform the above program into a λProlog program, one must declare the type of every constant, and adopt λProlog syntax (see section 2.1).

```
kind person type .
type (adam, eve, cain, abel, ...) person .
type (mother, father, ...) person -> person -> o .

father adam cain .      father adam abel .      ...
mother eve cain .       mother eve abel .       ...
```

Relations *parent* and *grand_father* can be defined as follows:

```
type (parent, grand_father) person -> person -> o .

parent Parent Child :- father Parent Child .
parent Parent Child :- mother Parent Child .

grand_father GFather GChild :- father GFather Parent , parent Parent GChild .
```

2.3.3 Introducing the existential quantifier

Up to now the difference with Prolog is only superficial. We introduce progressively the new features of λProlog.

One may already use an explicit quantification by observing that variable *Parent* in *grand_father* does not occur in the head of the clause. Then, it can be existentially quantified in the body of the clause.

```
grand_father GFather GChild :- sigma Parent\( father GFather Parent , parent Parent GChild ) .
```

The program *grand_father* is so small that the explicit existential quantification seems to buy nothing. What it buys in general is an accurate classification of variables and of their scope (see section 2.3.5).

2.3.4 Introducing implication

We now assume the data-base also contains a record of "presumed fathers".

```
type presumed_father person -> person -> o .
presumed_father enoch methuselah .      ...
```

We want to combine the presumed father and grand-father relationship into the notion of "presumed grand-father" in which presumed fathers are considered as fathers. There are two possibilities. First possibility, Prolog style, is to define relation *presumed_grand_father* on the model of relation *grand_father*.

```
type presumed_grand_father person -> person -> o .
presumed_grand_father PGFather PGChild :-
      ( presumed_father PGFather Parent ; father PGFather Parent ) ,
      presumed_parent Parent PGChild .
```

This is a bad idea because this definition *ex nihilo* does not show that relation *presumed_grand_father* contains relation *grand_father*. Moreover, the definition *ex nihilo* is not limited to relation *grand_father*; one must also build a relation for "presumed parent".

Second possibility is to reuse relation *grand_father* and assume that relation *father* contains relation *presumed_father*. So doing, the program structure displays the fact that relation *presumed_grand_father* contains relation *grand_father*.

```
presumed_grand_father PGFather PGChild :-
      ( (pi F\(pi C\(father F C :- presumed_father F C))) => grand_father PGFather PGChild ) .
```

Explicit quantifications of variables F and C at the assumption level are crucial. They play the same rôle as the quantifications that are implicit at the clause level in Prolog: they indicate that variables F and C must be renamed when solving a goal with the assumed clause. We could have omitted them for a different meaning:

```
presumed_grand_father_2 PGFather PGChild :-
      ( (father F C :- presumed_father F C) => grand_father PGFather PGChild ) .
```

In this case, variables F and C are implicitly quantified at the level of clause *presumed_grand_father_2*. They are not renamed when solving a goal with the assumed clause. This means that the assumed clause can only be used with one F and one C. In this application, it means the assumed clause can only be used once: A *presumed_grand_father_2* is a *grand_father* with only one presumptive link.

We can imagine explicitly quantifying only one of variables F and C. The definition of *presumed_grand_father* uses presumed fatherhood to characterize both the grandfather and the intermediary parent. If one wants to limit the use of presumed fatherhood to characterize the grandfather, it is enough to modify the quantifications as follows[4].

```
presumed_grand_father_3 PGFather PGChild :-
      (     (pi C\(father PGFather C :- presumed_father PGFather C ))
      =>    grand_father PGFather PGChild
      ) .
```

Variable *PGFather* is free in the assumed clause. Hence, the assumed clause will always be used with the same first argument: the presumed grandfather.

In Prolog, clause assertion is the closest correspondent to clause implication. However, there are two major discrepancies between the two. First, nothing forces Prolog to limit the life-time of an asserted clause to be a subproof. Second, there cannot be any free variable in an asserted clause in Prolog. Those that are free in the Prolog term representing the clause are automatically universally quantified in the clause that is actually asserted. For instance, asserting *(p X)* in Prolog results in adding clause *pi X\(p X)*, while implying *(p X)* in λProlog results in adding clause *(p X)*. Then, it is difficult, if not impossible, to specify logically in Prolog the subtleties of "presumed grand-fatherhood".

2.3.5 Introducing the universal quantifier

In the previous examples, we limited ourselves to defining new relations using already defined relations. For instance, relation *grand_father* is defined as a join of relations *father* and *parent*. Many other relations can be defined similarly. So, we want to abstract the process of defining relations as joins into a second-order relation. This relation could be used as follows:

```
grand_father GFather GChild :- join father parent GF , GF GFather GChild .
grand_mother GMother GChild :- join mother parent GM , GM GMother GChild .
paternal_grand_father PGFather GChild :- join father father PGF , PGF PGFather GChild .
```

A possible definition of *join* is as follows:

[4] In the Prolog style solution, one defines a new relation on the model of relation *grand_father*:

```
presumed_grand_father_3 PGFather PGChild :-
      ( presumed_father PGFather Parent ; father PGFather Parent ) , parent Parent PGChild .
```

```
type join (A->B->o) -> (B->C->o) -> (A->C->o) -> o .
% RJ is the join of R1 and R2
% ∀xy.[R_J(x,y) ⇔ ∃J.(R_1(x,J) ∧ R_2(J,y))]
join R1 R2 RJ :- pi x\(pi y\( RJ x y = sigma J\( R1 x J, R2 J y ) )) .
```

The universal quantifications specify that relation RJ must satisfy some intensional properties. They cannot help specifying extensional properties like that two relations have the same arbitrary graph. For instance, *pi x\(parent x cain => parent x abel)* is not provable. In other words, universal quantification in λProlog is not an enumeration over some domain. Even if in this example one may infer that x has type *person*, the proof of the universal goal will not try to replace it with every possible person (*adam*, *eve*, ...). Variable x is simply replaced by a new constant, distinct from every known person. If a proof is possible with this new constant, then it will be possible with every person.

In the example of relation *join*, and as opposed to relation *grand_father*, it is difficult to do without existential quantification. It avoids defining an intermediary predicate only for giving the proper scope to variable J. If one forgets *sigma J*, the clause would read $\exists J.(\forall xy.[R_J(x,y) \Leftrightarrow (R_1(x,J) \wedge R_2(J,y))])$ instead of $\forall xy.[R_J(x,y) \Leftrightarrow \exists J.(R_1(x,J) \wedge R_2(J,y))]$.

2.3.6 Introducing λ-terms

Instead of giving a logical definition of what is a join, one may give a functional definition.

```
join R1 R2 x\y\(sigma J\( R1 x J, R2 J y )) .
```

The example of relation *join* displays some symmetry between universal quantification and λ-abstraction. This symmetry is genuine even if it is not always as visible as in this example. It can be summed-up in the notion of *essentially universal quantification* that encompasses universal quantification in goals and λ-abstraction. The main theme of this article is to explore this symmetry, and to use it as a guide for programming.

2.4 More advanced examples

We present as more advanced examples the programming in λProlog of two elementary relations on lists: the concatenation of lists, and the transformation of lists into *function lists*. Function lists are specific of a new programming style that comes with λ-terms.

2.4.1 Concatenation of lists

Predicate *append* below implements a strongly typed variant of the concatenation relation.

```
type append (list T) -> (list T) -> (list T) -> o .
type append0 (list T) -> (list T) -> o .
type final (list T) -> o .

append0 [] L2 :- final L2 .
append0 [A|L1] [A|L3] :- append0 L1 L3 .

append L1 L2 L3 :- ( final L2 => append0 L1 L3 ) .
```

We have adopted an unusual programming to illustrate how a context can be transmitted through a program clause rather than through a term of the goal as is usual in Prolog.

In the usual definition (see page 4), the second argument is transmitted intact by parameter *L2* until it is used in the first clause. The rôle of *L2* is so special that most Prolog compilers can recognize this situation and treat it accordingly.

In the definition we propose, the second argument is transmitted to the base case through the added clause *(final L2)*.

Observing that *final* is used only in *append0*, we unfold its only occurrence and drop it. The new relation *append0* is defined recursively and without any base case, but every time it is called by *append* a specific base case is added to the program.

```
append0 [A|L1] [A|L3] :- append0 L1 L3 .
append L1 L2 L3 :- ( append0 [] L2 => append0 L1 L3 ).
```

This programming style makes it possible to avoid passing as parameters the constants of a relation (another example: the options and tables of a compiler).

2.4.2 Function lists

Predicate *list_flist* describes the correspondence between a regular list (for instance, *[1,2,3]* with type *(list int)*) and a *function list* (for instance, *z\[1,2,3|z]* with type *((list int)->(list int))*) that have the same elements.

The *functional* representation of incomplete data-structures (for instance, *difference lists*) has both a theoretical and practical interest.

The theoretical interest is that the expected, logical and operational semantics of functional representations coincide. It is not the case for their usual Prolog counterparts. We show the problem with Prolog using the example of difference lists.

A difference list is a pair of lists *L1-L2* which denotes the list *L* that satisfies *append(L,L2,L1)*. If the list *L2* is an unknown, then the concatenation of the lists denoted by *L1-L2* and *L3-L4* is the list denoted by the pair *L1-L4* if and only if *L2* and *L3* are made equal. Note that not every pair *L1-L2* is a difference list. For *L1-L2* to be a difference list, *append(L,L2,L1)* must hold for some *L*. For instance, *[1,2,3]-[4]* is not a difference list, but *[1,2,3]-[3]* and *[1,2|X]-X* are.

- Difference list *L-L* denotes the empty list because *append([],L,L)* is true for every *L*. However, with clause *(d_empty L-L)*, the goal *(d_empty [1|Z]-Z)* succeeds for every Prolog system that do not perform the occurrence-check (almost every system). Hence, there is no use for a logical representation of the empty list property.

- One may also derive from the definition that clause *(d_append L1-L2 L2-L4 L1-L4)* logically represents the concatenation relation. However, goal *(d_append L R [1])* (where *L* and *R* are unknowns) always succeeds, but does not bind either *L* or *R* to a difference list. Hence, there is no representation of concatenation that is both logical and non-directional.

- Clause *(d_double L LL :- d_append L L LL)* does not have the expected meaning:
  (d_double [1|X]-X [1,1,1|Y]-Y) succeeds if there is no occurrence-check,
  but *(d_double [1|X]-X [1,1|Y]-Y)* fails if there is one. So, even the directional representation of the concatenation relation is not covered by difference lists.

The functional representation does not have these problems. A functional list is a function *FL* with type *(list T)->(list T)* which denotes the list *L* that satisfies $\forall l.append(L,l,FL(l))$.

- From the above definition, one deduces that $z\backslash z$ denotes the empty list because $\forall l.append([],l,(z\backslash z\ l))$ holds. λProlog offers a complete and correct implementation of the logic of functional lists. For instance, clause *(f_empty z\z)* is a logical and usable implementation of the empty list property.

- It comes from the definition that clause *(f_append L1 L2 z\(L1 (L2 z)))* logically represents the concatenation of functional lists. For instance, goal *(f_append L R z\[1|z])* succeeds and produces non-deterministically the expected bindings for *L* and *R*: *[L←z\z]*, *[R←z\[1|z]]*, and *[L←z\[1|z]]*, *[R←z\z]*. Hence, a logical and non-directional representation of concatenation is possible.

- Clause *(f_double L LL :- f_append L L LL)* represents the expected relation: *LL* is *L* appended to itself without violating the occurrence-check.

The relation between a Prolog list and its functional encoding can be described in λProlog as follows:

```
type list_flist (list T) -> ((list T)->(list T)) -> o .
list_flist L FL :- pi l\( append L l (FL l) ).
```

The predicate reads as follows: *FL* is the functional version of a Prolog list *L* if for *every* list *l*, the concatenation of *L* and *l* is the application of *FL* to *l*. In this program, and as opposed to the program for relation *join*, one cannot simply replace the universal quantification by a λ-abstraction. However, it remains true that the universal quantification corresponds to the λ-abstraction in functional list *FL*.

The practical interest of functional representations is that they allow for a concise notation, in the term language, of the composition operations of a data-structure. If we have $CONC \stackrel{\text{def}}{=} \lambda r\backslash z\backslash (l\ (r\ z))$, and if *L1* and *L2* are functional lists, then *(CONC L1 L2)* denotes their concatenation. Indeed, term *CONC* is exactly the composition combinator of the λ-calculus. Many other useful combinators can be defined for functional lists: $NIL \stackrel{\text{def}}{=} z\backslash z$, $UNIT \stackrel{\text{def}}{=} e\backslash z\backslash [e \mid z]$ and $CONS \stackrel{\text{def}}{=} e\backslash \lambda z\backslash [e \mid (l\ z)]$.

The monoid structure can be encoded using these combinators [8]. Concatenation is a function, and no longer a relation. The price to pay is to represent the elements of the monoid using functional lists. It is easy to show that *CONC* and *NIL* satisfy the axioms of the monoid structure. One may even write these axioms in λProlog:

```
type monoid M -> (M->M->M) -> o .
% N and C are the neutral element and the constructor of a monoid
monoid N C :-
      pi m\( (C N m) = m , (C m N) = m ) ,                    % Neutral element
      pi m1\(pi m2\(pi m3\( (C (C m1 m2) m3) = (C m1 (C m2 m3)) ) .   % Associativity
```

3 Some incorrect intuitions

3.1 λProlog $\supseteq$ Prolog

That λProlog is defined by adding new capabilities to Prolog makes one think that λProlog *contains* Prolog. The most natural meaning one gives to "contains" is that

1. every Prolog program is a λProlog program, up to declarations and syntactic differences,
2. it retains its meaning.

Two counter-examples show that typing in λProlog invalidates these two propositions.

3.1.1 Some Prolog program have a different meaning in λProlog

That a Prolog program keeps its meaning in λProlog is the easiest proposition to invalidate. This is largely commented in the literature on prescriptive typing for Prolog [33, 16, 17, 24].

An example shows the problem. Predicate *append* as defined page 4 has the set $\{(append\ []\ x\ x) \mid x \in \mathcal{U}\}$ in its Prolog denotation, where $\mathcal{U}$ is the Herbrand universe. Hence, atom *(append [] 72 72)* is a logical consequence of the predicate.

With λProlog, constant *append* can only be used with a type that is an instance of its type scheme (e.g. the type declared page 4), $(list\ \gamma)\text{->}(list\ \gamma)\text{->}(list\ \gamma)\text{->}o$. This is also true of the denotation which must be computed in a typed logic. Hence, *(append [] 72 72)* is not a logical consequence of predicate *append* in λProlog.

In fact, this discrepancy between the denotations for the non-typed and typed versions of the same program often measures the gap that is observed between the expected semantics of a program and its actual denotation. For instance, the expected semantics of predicate *append* generally does not contain atom *(append [] 72 72)*.

3.1.2 Some Prolog programs are not λProlog programs

Predicate *flatten* relates a binary tree encoded by lists of lists to the lists of its leaves.

```
flatten [] [] :- ! .                                   % (list α)->(list β)->o
flatten [L|R] Lvs :- ! ,                               % (list α)->(list α)->o
      flatten L LLvs , flatten R RLvs , append LLvs RLvs Lvs .
flatten Lf [Lf] .                                      % α->(list α)->o
```

The program is annotated with the types of the occurrences of constant *flatten* in the head of the clause.

This predicate cannot be typed in λProlog. The reason is that *all the occurrences of a given predicate symbol in the head of a clause must have the same type* (up to names of type variable). This is not possible for this definition of *flatten*. We give lower another definition that can be typed.

The above condition is not clearly adopted in the definition of λProlog, but we believe it should be, and it is in our implementation of λProlog [6, 5]. This condition is already in the type system by Mycroft and

O'Keefe [33], but not discussed. It appears under the name *head condition* in works by Hanus, and Hill and Lloyd [16, 17]. It also appears under the name *definitional genericity* in Lakshman and Reddy's work [24].

According to generic polymorphism, types of all occurrences of *flatten* must be instances of its type scheme, but the head condition prevents from choosing a type scheme compatible with all the occurrences. Without the head condition, the type scheme α->β->o is compatible with every occurrence of *flatten*, but it is uninformative. With the head condition, the first clause forces the type scheme to be an instance of *(list* α*)->(list* β*)->*o, and the third clause forces it to be an instance of *(list* α*)->(list (list* α*))->*o. However, because the type scheme of *append* is *(list* γ*)->(list* γ*)->(list* γ*)->*o, the second clause enforces that α=*(list* α*)*, hence a contradiction.

We can see now that the expression

$$\lambda Prolog \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms + simple\ types + \ldots$$

is misleading. In fact, typing is not an additive component of λProlog; it is a filter that makes some configurations illegal.

Definitions such as *flatten* belong to the "art" of programming in Prolog [36]. They rely heavily on non-logical properties of Prolog (in this case, ordering of clause selection). Indeed, the third clause of *flatten* is not of the same kind as the others. It can be used only when the first argument is not a list, whereas the first two clauses can only be used when it is a list. The third clause is a default clause, the "else" branch of a conditional. This is a control structure that cannot be expressed in pure Prolog, and the non-logical cut, *'!'*, in the second clause is essential.

What can we do with a system that enforces the head condition? First, we should not use lists for representing binary trees, and more generally, we must ensure that every discrimination is done between constants of the same type, and not between constants of some type and terms of some other type. Then, one may safely write conditionals in Prolog by case analysis. A correct representation of binary trees is declared as follows[5]:

```
kind tree2 type -> type .
type leaf Lf -> (tree2 Lf) .
type node (tree2 Lf) -> (tree2 Lf) -> (tree2 Lf) .
```

In this context, relation *flatten* is defined as follows:

```
type flatten (tree2 Lf) -> (list Lf) -> o .
flatten (leaf Lf) [Lf] .
flatten (node L R) Lvs :- flatten L LLvs , flatten R RLvs , append LLvs RLvs Lvs .
```

There is no longer a conditional with a default branch[6].

This shows that some "classical" Prolog programs cannot be translated into λProlog by simply adding suitable declarations. The program we have used as an example shows the fundamental rôle that types play in λProlog as descriptors for data-structures.

3.2 Function terms

We have seen that some terms of λProlog (i.e. those with an arrow type) can be interpreted as functions. We have also seen that proofs in λProlog are done modulo $\alpha\beta$-equivalence. So, it looks like if one can program in a functional style in λProlog, and that unification is able to discriminate applications from abstractions and to analyse their components. We will show that these are two misconceptions about the capabilities of λ-terms in λProlog.

[5] If one needs labelling nodes, and not only leaves, the following declarations can be used instead:

```
kind tree2 type -> type -> type .
type leaf Lf -> (tree2 Lf Nd) .
type node (tree2 Lf Nd) -> Nd -> (tree2 Lf Nd) -> (tree2 Lf Nd) .
```

[6] The programming is willingly naïve. One may trivially eliminate the explicit concatenation.

3.2.1 Programming in a functional style in λProlog

One must recall that λProlog terms are essentially simply typed λ-terms. As such, they have a computing power that is too weak for really serving as a programming language.

ML programs are essentially made of simply typed λ-terms, but the language also offers a construction (i.e. *letrec*) that is interpreted by a fix-point combinator. That is what gives its computing power to ML.

What gives λProlog its computer power (which is the same as for any other reasonable programming language) is the structure of its clauses, but not its λ-terms. In other words, β-reduction in λProlog is not really used for evaluating functions. In fact, it is mainly used for instantiating term schemes represented by λ-terms.

However, λ-terms allow us to functionally encode operations that are sufficiently simple. Since variables are allowed in types, but the quantifications of these variables are always prenex, the domain of λProlog terms is essentially equivalent to ML$^-$ (ML minus *letrec*). This domain allows us to program extended polynomial functions only [19]. The functional lists example of section 2 shows how the concatenation function can be encoded into a λ-term.

3.2.2 Discriminating applications from abstractions in λProlog

A λ-term can be a constant, a variable, an abstraction, or an application. We will show that it is impossible to discriminate abstractions from applications, and to access their components without using a specific programming discipline.

In λProlog as in Prolog, the only means for discriminating terms and for accessing their subterms is unification.

A naïve solution to recognize that a term is an application and to access its components (the left and right members of the application) is to unify it with term $(T_1\ T_2)$ in order to bind its components to variables T_1 et T_2. In fact, it is much too naïve because term $(T_1\ T_2)$ is unifiable modulo $\alpha\beta$-equivalence with *any* term. For instance, terms 72, $(A\ B)$ and $x\backslash x$ are all unifiable with $(T_1\ T_2)$[7].

- Problem $72=(T_1\ T_2)$ has an infinite number of solutions, among which $[T_1 \leftarrow x\backslash x,\ T_2 \leftarrow 72]$ and $[T_1 \leftarrow x\backslash 72,\ T_2 \leftarrow 73]$.
- Problem $(A\ B)=(T_1\ T_2)$ has also an infinite number of solutions, among which $[T_1 \leftarrow x\backslash(x\ B),\ T_2 \leftarrow A]$.
- Problem $x\backslash x=(T_1\ T_2)$, too, has an infinite number of solutions, among which $[T_1 \leftarrow x\backslash x,\ T_2 \leftarrow x\backslash x]$.

In the case of term $(A\ B)$, the solution that allows us to actually access A and B is lost among the infinitely many others without any means for distinguishing it. Hence, the property of being unifiable with an application does not discriminate applications from other terms, and unification does not allow us to access to the components of applications.

Similarly, a too naïve solution to recognize that a term is an abstraction and to access its components (in fact only one, the body) is to unify it with term $v\backslash T$ in order to bind the body of the abstraction to variable T. This does not work because term $v\backslash T$, which does indeed only unify with abstractions, does not unify with every abstraction. The problem is that the substitution theory that underlies higher-order unification forbids capturing λ-variables outside the abstraction that bind them. However, this is precisely what we are trying to do by unifying an abstraction (say $x\backslash x$) with $v\backslash T$, and expecting that it will bind its body (here x) to T. In fact, the body of some abstraction can be substituted to variable T only if the body does not contain any free occurrence of the bound variable. The term $v\backslash T$ describes precisely those abstractions that represent constant functions. Hence, to be unifiable with $v\backslash T$ does distinguish only very particular abstractions.

One may compare these difficulties with well-known Prolog difficulties. The language of Prolog terms allows for variables in terms, however, there is no means in pure Prolog for checking that a term is a variable. In fact, the semantics of Prolog actually uses a saturation of the term domain, the Herbrand universe, in which there is no room for variable terms. Similarly, the semantics of λProlog uses λ-equivalence, which leaves no room for the application/abstraction discrimination.

The problem has two sides: to discriminate applications from abstractions, and to access their components. The solution for the first side is to label with a suitable constructor the applications and abstractions we want to be able to recognize. For instance, the representation of ML terms in λProlog can use the two following constructors:

[7] $(T_1\ T_2)$ is not simultaneously unifiable with 72, $(A\ B)$ and $x\backslash x$, but three instances of $(T_1\ T_2)$ with three different types are unifiable with 72, $(A\ B)$ and $x\backslash x$.

```
kind mlt type .
type app mlt -> mlt -> mlt .          % Application: write (app F X) instead of (F X)
type fun (mlt->mlt) -> mlt .          % Abstraction: write (fun x\x+1) instead of x\x+1
```

Using this technique, one can easily discriminate abstractions from applications. One can also access the components of some discriminated application. The problem of accessing the body of an abstraction is solved in the next section.

4 The reconstruction of λProlog

We show how adding λ-terms to Prolog leads to adding every other λProlog component. The central idea that links all the components is the notion of essentially universal quantification. It comes from the analysis of inductive programming on data-structures.

4.1 Adding λ-terms

4.1.1 Motivation

The motivation to manipulate λ-terms in logic programming is simple: they are the most natural representation for structures that feature either scoping, compositionality, or some form of genericity [26].

Scoping, compositionality and genericity are not exclusive. Among the structures that feature scoping, we see logical and mathematical formulas (quantifications: $\forall u$, $\exists v$, ...; sums and products: $\sum_{x \in X} x$, $\prod_{i \in I} x_i$, $\int_0^1 f(x).dx$, ...; derivations: d/dx, $\partial/\partial x$, ...), and computer programs (parameterisation: *f(x) int x; { ... }* , ...; blocks: *{ int x; ... }* , ...). Among those that feature compositionality, we see expressions of compositional semantics (denotational: $\mathcal{T}_g[\![B_1, B_2]\!] = \lambda\kappa.(\mathcal{T}_g[\![B_1]\!]\ (\mathcal{T}_g[\![B_2]\!]\ \kappa))$ [7], ...; Montague semantics for natural language [10]). Finally, logical quantifications in automated demonstration feature a form of genericity: one instantiates them using substitutions for building proofs.

The following table pictures some possible representations using λ-terms.

$\forall u.P(u)$	*forall u\(P u)*
$\sum_{x \in X} x$	*sum X x\x*
$\int_0^1 f(x).dx$	*integral 0 1 x\(f x)*
df/dx	*derivative x\(f x)*
f(x) int x; { ... }	*function int x\(f x) ...*
{ int x; ... }	*block int x\...*
$\mathcal{T}_g[\![B_1, B_2]\!] = \lambda\kappa.(\mathcal{T}_g[\![B_1]\!]\ (\mathcal{T}_g[\![B_2]\!]\ \kappa))$	*t_g (B1 and B2) k\(D1 (D2 k)) :- t_g B1 D1, t_g B2 D2*

Scoping introduces the notion of *scoped variable* or *parameter*. The key operation for composing structures is the *substitution* of a term to a parameter; it must be sound with respect to scoping. Axiom β models such a substitution. This is why λ-terms are well suited for representing these structures: abstraction serves as a generic quantification.

All these structures can be represented with first-order terms, but the correct handling of substitution with respect to scoping needs a lot of attention. The most frequent solution is to represent the parameters (the *object-level variables*) with Prolog variables (the *meta-variables*). There are numerous examples of this solution in text-books. They can be found in chapters dedicated to the manipulation of programs and formulas.

We take as an example the following clause from program 16.2 (page 259) in *The Art of Prolog* by Sterling and Shapiro [36][8].

```
translate( (A,B), (A1,B1), Xs/Ys ) :- translate( A, A1, Xs/Xs1 ) , translate( B, B1, Xs1/Ys ) .
```

This clause is part of a program for translating grammar rules into Prolog. Every clause of predicate *translate* has the same structure. The first parameter is a component of some grammar rule, the second is the corresponding component in the target Prolog clause, and the last one is a pair of variables that must occur in the target Prolog clause. So, these two variables are object-level variables. The problem is that it is no longer possible to give a declarative reading to this clause because the object-level variables are represented directly with

[8] This is an almost verbatim copy of the clause. The only modifications are for avoiding clashes with notations used in this article. In the original text, ':-' is noted ←, and '/' is noted '\'.

meta-variables. The declarative reading of Prolog does not spell out the actual rôle of the X's and Y's. They are variables of the target clause, and it makes no sense to consider instance of them as the declarative semantics of Prolog does. It works in Prolog because the interpreter always computes the most general solutions. So, it is the operational semantics than gives the meaning of this clause, instead of the declarative semantics.

In the same book, clause *derivative(X,X,s(0))* (program 3.29, page 63) is part of a program that computes the derivative with respect to X of a function. One of its logical consequences is *derivative(72,72,s(0))*, which is absurd. Clause *polynomial(X,X)* (program 3.28, page 62) is a part of the definition of what is a polynomial in X. It has also absurd logical consequences: *polynomial(72,72)*.

In Prolog, substitution of object-level variables is easy at the price of declarativity. Then, it forces the programmer to check the correctness of object-level terms manipulations with respect to the operational semantics.

Using λ-terms permits a coherent and declarative handling of scopes and substitutions.

4.1.2 Adding $\alpha\beta$-equivalence

To add λ-terms to Prolog is still a fuzzy objective; one must decide the circumstances.

A first observation is that there is no use in adding λ-terms without $\alpha\beta$-equivalence. Indeed, it is $\alpha\beta$-equivalence that allows us to substitute terms to variables with respect to scoping. So, system $Prolog_\lambda \stackrel{\text{def}}{=} Prolog + \lambda\text{-}terms$ without $\alpha\beta$-equivalence is useless.

In order to add λ-terms and $\alpha\beta$-equivalence to Prolog, one must also add the handling of axioms α and β to unification. One also wants to be sure that there is always a normal form to λ-terms. The domain of simply typed λ-terms has been selected because it is strongly normalizable. The unification problem for simply typed λ-terms modulo axioms α and β is semi-decidable and infinitary[9]. In practice, one uses the semi-algorithm proposed by Huet [20].

4.2 Consequence of adding λ-terms

We have added simply typed λ-terms and $\alpha\beta$-equivalence to Prolog. Does this result in a usable logic programming language? The answer is no.

We have shown in section 3 that unification (even higher-order) alone does not allow us do discriminate abstractions from applications or to access their components. Having added simply typed λ-terms and $\alpha\beta$-equivalence to Prolog, we are able to build and compare them, but we cannot analyse them. In other words, we cannot perform an inductive traversal of the terms we are able to build.

The difficulty with abstractions comes from the fact that it is not possible to capture free λ-variables in bindings of logical variables.

The intuition of the λProlog solution is that to access the body of an abstraction A, the only means is to apply A to a term t and use term *A'=(A t)*. Knowing *A'* and t, one must be able to compute A by solving *A'=(X t)* for an unknown X. So, the equality theory of the λ-terms and the term t must be such that given two distinct terms A and B, the terms *(A t)* and *(B t)* are also distinct. We call this condition the *conservation condition*. It ensures that accessing the body of an abstraction is a reversible operation.

We show in next sections that the conservation condition is satisfied if and only if,

1. the equality theory of λ-terms entails η-equivalence,
2. the term t is recognizable (in a sense that will be made clear in another section).

4.2.1 Axiom η

To apply an abstraction to a term in order to access its body does not allow us to discriminate between η-equivalent terms (e.g. E and $x\backslash(E\ x)$). Indeed, $(x\backslash(E\ x)\ t) =_{\alpha\beta} (E\ t)$ even if $x\backslash(E\ x) \neq_{\alpha\beta} E$.

Hence, the conservation condition implies η-equivalence. So, $\alpha\beta$Prolog (i.e. λProlog without η-equivalence) is not a usable logic programming system.

4.2.2 Universal quantification in goals : $\forall_\mathcal{G}$

We have seen that we plan to analyse an abstraction A by applying it to some term t and analysing the result *A'=(A t)*.

[9] There can be infinitely many most general unifiers.

If one wants to be able to compute A by solving $A'=(X\ t)$ for an unknown X, it must be that t is recognizable among the subterms of A'. As a counter-example, if $A=x\backslash(x+72)$ and $t=72$, we get $A'=(72+72)$. The equation $(72+72)=(X\ 72)$ has four solutions among which there is no formal reason to prefer one: $[X\leftarrow x\backslash(x+x)]$, $\underline{[X\leftarrow x\backslash(x+72)]}$, $[X\leftarrow x\backslash(72+x)]$, and $[X\leftarrow x\backslash(72+72)]$. The four solutions correspond to four A's that are not $\alpha\beta\eta$-equivalent but still yield the same term $(A\ t)$ for $t=72$. The underlined solution is the one we informally prefer. We need some formal means to select this one.

One could object that it was really gross to choose $t=72$ when A already has 72 as a subterm. This objection does not hold for two reasons. First, because a term A can have unknown subterms, it is not always possible to check that a candidate t does not occur in A. Second, even when the term A is fully determined (ground), to choose a t that does not occur in it does not completely solve the problem. For instance, if $A=x\backslash(x+72)$ and $t=73$, we get $A'=(73+72)$. Equation $(73+72)=(X\ 73)$ has two solutions among which there is still no formal means for selecting one: $\underline{[X\leftarrow x\backslash(x+72)]}$ and $[X\leftarrow x\backslash(73+72)]$. Again, two non-$\lambda$-equivalent A's yield the same $(A\ t)$ for $t=73$. In fact, for any equation $(A\ t)=(X\ t)$ in X, and if t is an ordinary term, there are always at least solutions $[X\leftarrow x\backslash(A\ t)]$ and $\underline{[X\leftarrow A]}$.

Something must prevent term t from occurring in A and in X. This will bar all the solutions that are not underlined. In other words, it provides the formal means we want for preferring the underlined solutions.

Universal quantification in goals, $\forall_{\mathcal{G}}$, is such a logical means for producing a recognizable term t. If t is universally quantified in the scope of the quantifications of A and X, then the only solution of equation $(A\ t)=(X\ t)$ is $\underline{[X\leftarrow A]}$. The goal to prove has the following form: $\exists A \exists X \forall t.\ (A\ t)=(X\ t)$. Because of η-equivalence, this goal is equivalent to $\exists A \exists X.\ A=X$, which has the trivial solution $\underline{[X\leftarrow A]}$[10].

This shows a fundamental correspondence between abstraction and universal quantification. They cannot be considered independently. Informally speaking, abstraction is an essentially universal quantification operating at the term level. Abstraction is a storable/manipulable form of universal quantification, and universal quantification is the way for interpreting abstraction.

4.2.3 Implication in goals : $\Rightarrow_{\mathcal{G}}$

We have seen that the universal quantification allows us to interpret abstractions. We also have seen in section 2 that the deduction system interprets universal quantifications by substituting a *new* constant to the universal variable. The constant is simply added to the current signature.

This new constant is a problem: how can the programmer take it into account? It is trivial that, because the constant is new, no predicate definition can take it into account. Predicates that are supposed to be defined for every constructor of a data-structure are no longer completely defined when a new constant is introduced.

We need some means for augmenting a predicate definition during the life of a new constant: i.e. during a subproof. Implication in goals, $\Rightarrow_{\mathcal{G}}$, is such a means because it allows us to augment the program for the same life-time.

4.2.4 Summing-up

This part can be summed-up as follows: for adding λ-terms and $\alpha\beta$-equivalence to Prolog, one needs to type terms in a discipline that makes the unification problem well-defined. For defining relations by structural induction on abstractions, and still satisfy some conservation condition, one must also add η-equivalence and universal quantification in goals. To be able to maintain the completeness of the inductive definitions in presence of universal variables (new constants), one needs to add implication in goals.

Next section gives programming examples that illustrate this scheme.

5 Programming by structural induction

The practical interest of our reconstruction of λProlog is that it provides a guide for programming. We present two examples: a predicate that relates a first-order predicate calculus formula to its negative normal form[11], and a predicate that relates a pure λ-calculus term to its simple type.

[10] Other most general solutions exist, but they are renaming of this one.

[11] A first-order predicate calculus formula is in negative normal form if the negation connective is only applied to atomic formulas. For instance, $(\neg A)\vee(\neg B)$ is in negative normal form, but $\neg(A\wedge B)$ is not. It is always possible to transform a first-order predicate calculus formula into a negative normal formula using De Morgan's identities.

5.1 Defining data-structures

The first thing to do is to define data-structures: i.e. types and terms constants.

5.1.1 Formulas of the first-order predicate calculus

We need two types for representing the formulas of the first-order predicate calculus: a type for truth values, and a type for individuals. They are *object-level* truth values and individuals. They are represented by λProlog λ-terms, but they must not be mistaken with λProlog formulas or terms.

```
kind (formula, individual) type .
```

One also needs connectives for object-level formulas.

```
type (and, or, impl) formula -> formula -> formula .
type not formula -> formula .
type (forall, exists) (individual->formula) -> formula .
type (p, ...) individual -> individual -> formula .
type (q, ...) formula .
```

So, formula $\forall x.(p(x,x) \Rightarrow q)$ is encoded by λ-term *(forall x\(impl (p x x) q))*.

The only constructors that are original with respect to Prolog are *forall* and *exists*. They have an argument that is an abstraction of λProlog. Its rôle is to formalise the scope of object-level quantifications and to handle the fact that quantified variables are substitutable.

The semantics of object-level connectives must be defined by deduction rules, axioms, etc, written in λProlog.

5.1.2 Simply typed λ-terms

One also needs two types for representing the terms of the simply typed λ-calculus: a type for λ-terms and another for simple types. As for the representation of object-level formulas of the predicate calculus, they must not be mistaken for the meta-level λ-terms and types.

```
kind (l_term, simple_type) type .
```

One needs constructors for representing object-level applications and abstractions, the arrow of object-level simple types, and various object-level type and term constants.

```
type app l_term -> l_term -> l_term .
type abs (l_term->l_term) -> l_term .
type arrow simple_type -> simple_type -> simple_type .
type (one, two, three, ...) l_term .
type (integer, real, ...) simple_type .
```

The only constructor that is original with respect to Prolog is *abs*. Its argument is an abstraction of λProlog whose rôle is to formalise the scope of the object-level abstraction and the fact that the bound variable is substitutable.

It is important to understand that the λProlog abstraction in the representation does not model all the semantics of the object-level abstraction. For instance, *(app (abs E) F)* $\neq_{\alpha\beta}$ *(E F)*. If such relation holds at the object level it must be described in λProlog by a suitable predicate:

```
type beta_conv l_term -> l_term -> o .
beta_conv (app (abs E) F) (E F) .
```

In this clause, the meta-level β-reduction of *(E F)* performs the actual substitution of term *F* to the variable bound by *(abs E)*.

5.2 Defining properties

For using a property of an object-level structure, it must be explicitly defined in λProlog beforehand. When possible, it is defined by an induction on the constructors of its λ-terms representation. If the type is *inductive* then it is easy to deduce an induction function on the constructors [4][12].

Let us show that type *l_term* is not inductive. Its constructors are *app* and *abs*, and the type of *abs* is *(l_term->l_term)->l_term*. Then, the type of the only argument of *abs* is *l_term->l_term*. It follows *l_term* has a negative occurrence in the type of an argument of one of its constructors (*neg(l_term->l_term)*={*l_term*}); then it is not inductive.

Let us show that type *formula* is inductive. Its constructors are either like *and*, or like *forall*. Every argument of *and* has type *formula*. The only argument of *forall* has type *individual->formula*. Because *formula* does not occur negatively in them (*neg(formula)*=∅ and *neg(individual->formula)*={*individual*}) it is an inductive type.

In λProlog, we can use a kind of induction even when the type of a data-structure is not inductive.

5.2.1 Induction on formulas

We describe by structural induction the relation between a first-order predicate calculus formula and its negative normal form.

We first declare the type of the relation.

```
type nnf formula -> formula -> o .
```

Now we define relation *nnf* by structural induction on type *formula*. The cases for all the connectors are elementary and could be described in Prolog.

```
nnf (and F1 F2) (and G1 G2) :- nnf F1 G1 , nnf F2 G2 .
nnf (or F1 F2) (or G1 G2) :- nnf F1 G1 , nnf F2 G2 .
nnf (not (and F1 F2)) (or G1 G2) :- nnf (not F1) G1 , nnf (not F2) G2 .
nnf (not (or F1 F2)) (and G1 G2) :- nnf (not F1) G1 , nnf (not F2) G2 .
nnf (p A B) (p A B) .
nnf (not (p A B)) (not (p A B)) .                                   % nnf1
...
```

The cases for quantifications is more interesting. One must continue the induction in the quantified formula. Then a universal quantification is required. It introduces a universal constant of type *individual*, which is not the induction type. Then, augmenting the inductive definition for this new constant is not required.

```
nnf (forall F) (forall G) :- pi i\( nnf (F i) (G i) ) .
nnf (exists F) (exists G) :- pi i\( nnf (F i) (G i) ) .              % nnf2
nnf (not (forall F)) (exists G) :- pi i\( nnf (not (F i)) (G i) ) .
nnf (not (exists F)) (forall G) :- pi i\( nnf (not (F i)) (G i) ) .  % nnf3
```

A few observations are in order.

- If the object-level theory had been a higher-order logic in which variables of type *formula* can be quantified, then type *formula* would not have been inductive, and it would have been required to augment the inductive definition using λProlog implication.
- In relation *nnf*, we use λProlog universal quantification in a way that has nothing to do with the semantics of object-level formulas. It has only to do with their structure. The two object-level quantifications, *exists* and *forall*, presumably respectively existential and universal, are interpreted by a universal quantification.

[12] One says the type of a data structure is inductive if its constructors admit arguments of that same type only in positive occurrences [35].

We recall that following the *Curry–Howard isomorphism* [11, 18] the arrow of simple types is analogous to implication in propositional intuitionistic calculus. As does implication, arrow introduces a notion of positive and negative occurrences as follows:

$$pos(A\text{->}B) \stackrel{\text{def}}{=} neg(A) \cup pos(B)\ , \quad neg(A\text{->}B) \stackrel{\text{def}}{=} pos(A) \cup neg(B)$$
$$pos(T) \stackrel{\text{def}}{=} \{T\}\ , \quad neg(T) \stackrel{\text{def}}{=} \emptyset \qquad \text{if } T \text{ is not an arrow type}$$

For instance, $pos((a\text{->}b)\text{->}(c\text{->}d))=\{a,d\}$ and $neg((a\text{->}b)\text{->}(c\text{->}d))=\{b,c\}$.

- When we use predicate *nnf* for normalizing a formula, the universal quantification works both in analysing and in synthesising abstractions. Abstraction F is analysed and its body passed as a parameter to the recursive call to *nnf*. The second argument is unified with a term *(exists G)* or *(forall G)* where G is an unknown. The application of G to i is also passed as a parameter to the recursive call to *nnf*. The unknown G being quantified out of the scope of the i, it cannot have any occurrence of i in its binding values. Hence, it will be bound to an abstraction that will become more and more precise as long as formula F is traversed.

We present the processing of an actual proof for illustrating the last observation.

1. The question.
   (nnf (exists x\(not (exists y\(p x y)))) X)
2. Resolution with clause nnf_2 of *nnf* (page 18): *[X←(exists G)]*.
   pi i\(nnf (not (exists y\(p i y))) (G i))
3. Rule $\forall_{\mathcal{G}}$ (page 6).
   (nnf (not (exists y\(p c y))) (G c))
4. Resolution with clause nnf_3 of *nnf*: *[G←x\(forall (H x)), G'←(H c)]* is the most general solution to *(G c)=(forall G')* [20].
   pi i\(nnf (not (p c i)) (H c i))
5. Rule $\forall_{\mathcal{G}}$.
   (nnf (not (p c c')) (H c c'))
6. Resolution with clause nnf_1 of *nnf*: *[H←x\y\(not (p x y))]*.
   Success
7. The solution is *[X←(exists x\(forall y\(not (p x y))))]*.

5.2.2 Induction on simply typed λ-terms

The well-typing relation is defined by structural induction on object-level terms.

The type of the well-typing relation is as follows:

```
type typing l_term -> simple_type -> o .
```

Then it is defined by induction on the constructors of type *l_term*. The case for applications is elementary and could be written in Prolog.

```
typing (app T1 T2) B :- typing T1 (arrow A B) , typing T2 A .
```

The case for abstractions is more interesting. The induction through object-level abstractions uses a universal quantification because object-level abstractions are represented by λProlog abstractions. This universal quantification introduces a universal constant of type *l_term*, which is the type on which the induction operates; this is because *l_term* is not an inductive type. So, one must augment the inductive definition for the life-time of the universal constant using an implication.

```
typing (abs E) (arrow A B) :- pi x\( typing x A => typing (E x) B ) .
```

The added clause is *(typing x A)*. It contains a free logical variable A. This forces all the occurrences of constant x to have the same type. If the added clause had been *pi T\(typing x T)*, every occurrence would have had its own type.

The above sequence of reasoning provides only the structure of the λProlog program. To work out the full details needs to know the logic of the defined relation. In this case, the logic is given by the deduction rules of the theory of simple types [2, 19]. For the abstraction, the rule is called *arrow introduction*:

$$\frac{\Gamma, x : \alpha \quad \vdash \quad E : \beta}{\Gamma \quad \vdash \quad \lambda x.E : \alpha \rightarrow \beta} \qquad \rightarrow_I$$

5.3 A programming scheme for induction on types

We present a general scheme for programming by induction on types.

The constructors that allow us to discriminate applications from abstractions must be declared as follows:

type c_{app} τ_1 -> τ_2 -> τ_0 .
type c_{abs} (τ_3->τ_4) -> τ_0 .

where τ_0 is the type in which one wants to discriminate applications from abstractions. In every concrete use of this scheme, the constructors may have supplementary arguments, and there can be several constructors of the two kinds.

Constructor c_{app} is used to tag applications. Its type contains the types of the two components of an application. They cannot simply be juxtaposed in an application; one could not recognize them. So, they must be two separate arguments of c_{app}.

Constructor c_{abs} helps distinguishing abstractions. One finds in its type the type of the abstraction, which is simply an argument of c_{abs}. Note that "the type of the abstraction" is not the type of the object-level abstraction; it is the type of the λProlog abstraction that represents an object-level scoped construct.

Any induction on the constructors of type τ_0 must be written as follows:

type induction$_0$ τ_0 -> *o* .
induction$_0$ (c_{app} *Term*$_1$ *Term*$_2$) :- *induction*$_1$ *Term*$_1$, *induction*$_2$ *Term*$_2$.
induction$_0$ (c_{abs} *Abstr*) :- *pi* t_3\(*induction*$_4$ (*Abstr* t_3)) .

Predicates *induction*$_i$ are defined by inductions on types τ_i. Implicitly, universal variable t_3 has type τ_3. Such a scheme works in every mode: term *Abstr* can be either analysed or synthesised during a proof that uses this clause. For a concrete use of the scheme, relations *induction*$_i$ may have supplementary arguments. For instance, relation *typing* has a second argument that is an object-level type (see page 19).

Quantification "*pi* t\" is in the scope of the (implicit) quantification of *Abstr*. It can be made explicit as follows:

pi Abstr\(*induction*$_0$ (c_{abs} *Abstr*) :- *pi* t_3\(*induction*$_4$ (*Abstr t*))) .

If relation *induction*$_3$ is used in relation *induction*$_0$ (i.e. types τ_0 and τ_3 are not mutually inductive[13]), one must augment the definition of *induction*$_3$ for encompassing the universal variable t_3. So, one finds often the following idiom:

type induction$_0$ τ_0 -> *o*.
induction$_0$ (c_{abs} *Abstr*) :- *pi* t_3\(*induction*$_3$ t_3 :- ... => *induction*$_4$ (*Abstr* t_3)).

In a concrete use of the scheme, relation *induction*$_3$ may have supplementary arguments. These arguments and the body of the new clause may contain free occurrences of logical variables. For instance, in relation *typing* (page 19), variable *A* is free in the added clause. Conversely, the added clause may have its own universal quantifications: they must be explicit as in relation *presumed_grand_father* (page 8).

6 Conclusion

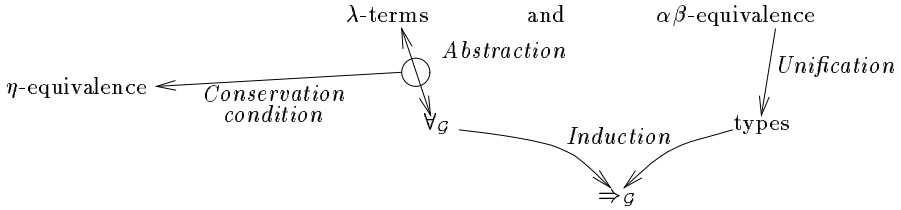
We have presented a reconstruction of λProlog based on the necessity relationships its various features entertain. We also have exhibited a second essentially universal quantification in λProlog: abstraction. It matches at the term level universal quantification in goals. This symmetry gives the hints of a method for programming in λProlog: to abstractions in data-structures must correspond universal quantifications in programs.

6.1 The structure of λProlog

6.1.1 A summary

The following picture illustrates the relationships that give a structure to the features of λProlog. Arrows read "requires".

[13] The notion of inductiveness extends easily to types defined by mutual recursion.



We consider λ-terms and $\alpha\beta$-equivalence as the principal components of λProlog, those that draw every other component. Adding these components to Prolog is motivated by the need for a more declarative handling of notions such as scoping and substitution. Then come types for making unification well-defined and tractable, and universal quantification in goals for handling induction through abstractions. Then η-equivalence in the equality theory is required for making the interpretation of abstraction by universal quantification correct and reversible. Finally, for inductive definitions to remain complete with respect to types when universal quantification introduces new constants, implication is needed for extending the inductions for the new constants.

Our reconstruction does not deal with disjunction in goals, $\vee_{\mathcal{G}}$, and with existential quantification in goals, $\exists_{\mathcal{G}}$. They do not really belong to Horn formulas, but their right-introduction rules are representable by second-order Horn clauses.

```
type ';' o -> o -> o .        type sigma (T->o) -> o .

G1 ; G2 :- G1 .       G1 ; G2 :- G2 .                    % P ⊢ G1 / P ⊢ G1∨G2   ∨G

sigma G :- (G T) .                                       % P ⊢ (G t) / P ⊢ ∃x.G   ∃G
```

So, they are not difficult at all, and most Prolog systems feature $\vee_{\mathcal{G}}$. They are mostly useful for building complex formulas without using intermediary predicates.

Among the fragments of λProlog we have presented in the introduction, we have seen that CLP($\lambda_{\rightarrow}$) and $\alpha\beta$Prolog are useless because they feature components that are high in the diagram without featuring the components that are below them (i.e. that come as a consequence). We analyse now the status of Typed Prolog and Harrop Prolog. Typed Prolog seems a good approach to programming in first-order λProlog, because this dialect invites the programmer to describe the data-structures more precisely than what is usually done in Prolog. It is an important point of the proposed programming method. The language Harrop Prolog seems to be only a "thought experiment" because it offers connectives $\forall_{\mathcal{G}}$ and $\Rightarrow_{\mathcal{G}}$, but does not feature abstraction for matching them. Moreover, we have seen that higher-order terms are useful for representing quantified formulas (e.g. for meta-programming). Prolog avoids higher-order terms by using unsound techniques (representing object-level variables by meta-variables) that may work when the quantification structure is simple, as in Horn formulas, but are not practicable when it is as complex as with hereditary Harrop formulas.

6.1.2 The structure of a restriction of λProlog

Miller proposes a fragment of λProlog, L_λ, that restricts more strongly the term domain than simple types do. In this fragment, the unification problem is decidable and unitary [28]. The term domain of L_λ is the greatest subset of λ-terms for which β_0-equivalence (defined below) is equal to β-equivalence.

β_0 — $(x\backslash E\ x) =_{\beta_0} E$.

> This axiom formalises a weak form of β-equivalence for the case the actual parameter is a variable. β_0-Reduction amounts to variable renaming.

The subdomain of the λ-terms for which β_0-equivalence is complete with respect to β-equivalence is described by a restriction of the rule for building applications. The only allowed applications with a variable head are those whose head is applied to distinct essentially universal variables.

Despite its algorithmic interest, the L_λ fragment is not widely used because lots of useful definitions do not belong to L_λ. For instance, the definition of *sigma* above does not belong to L_λ because of term *(B _t)*. Relation *mapfun* is not in L_λ either.

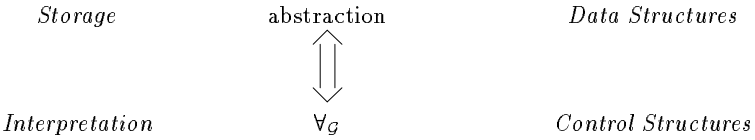
```
type mapfun (A->B) -> (list A) -> (list B) -> o .
mapfun F [] [] .
mapfun F [E|L] [(F E)|FL] :- mapfun F L FL .        % (F E) is not in Lλ
```

However, the L_λ fragment can be used as a heuristic criterion to avoid using the general but costly unification procedure [6, 5].

One may wonder if the L_λ fragment is restricted enough for modifying the overall structure of the language. In fact, L_λ has exactly the same structure as λProlog. Unification in L_λ still does not allow us to analyse every data-structure. The restriction is such that even application *(T1 T2)* cannot be formed in L_λ. So, one still needs universal quantification, η-equivalence, and implication.

6.2 Idioms for programming

The symmetry between abstraction and universal quantification in goals makes us consider the first as an essentially universal quantification. This is an important point for programming. This duality is often the cause of a conjoint use of abstraction and universal quantification in goals. It may be pictured as follows:



Abstraction is a *reified* form of universal quantification, the latter being a *reflection* of the former. For analysing/consuming/reflecting an abstraction, it is required to apply it to a universal variable whose scope is included in the scope of the variable that is bound to the abstraction. For synthesising/producing/reifying an abstraction, it is required to build a term that represents its body in which a unique universal variable represents every occurrence of the variable bound by the abstraction, and compute the function that yields that term when it is applied to the same universal variable (see for instance the proof of *(nnf (exists x\(not (exists y\(p x y)))) X)*, page 19).

One can deduce from our pragmatic reconstruction of λProlog a methodology for building programs by induction on a data-structure. This methodology allows us to decide where to use universal quantification and implication in goals, by simply looking at the types of the data-structure constructors.

6.2.1 Other idioms

This article does not cover every way of programming in λProlog. λProlog programming with first-order terms often uses techniques similar to Prolog's. However, there are new possibilities, such as used in predicate *append* of section 2.4.1. This predicate is an example of another idiom in which implication is used to make some term *global*. Another introductory example, function lists, presents another important idiom: functional data-structures. They play in λProlog a rôle similar to incomplete data-structures in Prolog, but they allow for a more declarative programming.

6.3 λProlog's status

We finish with a few words on the applications and availability of λProlog.

Possible applications are chiefly the applications that motivated the λ-terms: manipulation of formulas, computation of denotation, etc (see section 4.1). Among actual applications, we find automatic theorem proving [3, 14], analysis of natural and formal languages [34, 12, 21], and the manipulation of functional programs [15]. Notice also that the structure of λProlog encompasses such constructions as modules [29] and abstract data types [27] without any extra-logical addition.

Two implementations of λProlog can be used: one, called eLP, is an interpreter written in Lisp [13], the other, called Prolog/Mali, is a compiler written in λProlog which generates C programs [5]. A third implementation is in progress [23].

References

[1] H. Andreka and I. Nemeti. *The Generalised Completeness of Horn Predicate-Logic as a Programming Language*. DAI Research Report 21, University of Edinburgh, 1976.

[2] H. Barendregt. Introduction to generalized type systems. *J. Functional Programming*, 1(2):125–154, 1991.

[3] C. Belleannée. *Vers un démonstrateur de théorèmes adaptatif*. Thèse, Université de Rennes I, 1991.

[4] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992. ftp: //ftp.irisa.fr/local/lande.

[6] P. Brisset and O. Ridoux. *The Compilation of λProlog and its execution with MALI.* Technical Report 687, IRISA, 1992. ftp: //ftp.irisa.fr/local/lande.

[7] P. Brisset and O. Ridoux. Continuations in λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 27–43, MIT Press, 1993.

[8] P. Brisset and O. Ridoux. Naïve reverse can be linear. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 857–870, MIT Press, 1991.

[9] J. Cohen. Constraint logic programming languages. *CACM*, 33(7):52–68, 1990.

[10] S. Coupet-Grimal. Représentation sémantique dans le traitement des langues naturelles en Prolog. In *Journées Francophones sur la Programmation en Logique*, pages 69–91, Teknea, Nîmes, France, 1993.

[11] H.B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, 1968.

[12] M. Dalrymple, S.M. Shieber, and F.C.N. Pereira. Ellipsis and higher-order unification. In *Linguistics and Philosophy*, pages 399–452, 1991.

[13] C.M. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325, MIT Press, 1991.

[14] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *J. Automated Reasoning*, 11(1):43–81, 1993.

[15] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 4(2):415–459, 1992.

[16] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[17] P.M. Hill and J.W. LLoyd. *The Gödel Report.* Technical Report TR–91–02, University of Bristol, 1991.

[18] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, London, 1980.

[19] G. Huet. Introduction au λ-calcul typé. In B. Courcelle, editor, *Logique et informatique : une introduction*, pages 137–162, INRIA, 1991.

[20] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[21] S. Le Huitouze, P. Louvet, and O. Ridoux. Logic grammars and λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 64–79, MIT Press, 1993.

[22] J. Jaffar and J.-L. Lassez. *Constraint Logic Programming.* Technical Report 86/74, Monash University, Victoria, Australia, June 1986.

[23] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 871–886, MIT Press, 1991.

[24] T.K. Lakshman and U.S. Reddy. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp.*, pages 202–217, 1991.

[25] J.W. Lloyd. *Foundations of Logic Programming. Symbolic computation — Artificial Intelligence*, Springer-Verlag, Berlin, FRG, 1987.

[26] D.A. Miller. Abstract syntax and logic programming. In A. Voronkov, editor, *2nd Russian Conf. Logic Programming, LNCS 592*, Springer-Verlag, 1991.

[27] D.A. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *6th Int. Conf. Logic Programming*, pages 268–283, MIT Press, 1989.

[28] D.A. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.

[29] D.A. Miller. A proposal for modules in λProlog. In R. Dyckhoff, editor, *Int. Workshop Extensions of Logic Programming, LNAI 798*, pages 206–221, Springer-Verlag, 1993.

[30] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming, LNCS 225*, pages 448–462, Springer-Verlag, 1986.

[31] D.A. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[32] D.A. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In D. Gries, editor, *2nd Symp. Logic in Computer Science*, pages 98–105, Ithaca, New York, USA, 1987.

[33] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[34] R. Pareschi and D.A. Miller. Extending definite clause grammars with scoping constructs. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, pages 373–389, MIT Press, 1990.

[35] B. Pierce, S. Dietzen, and S. Michaylov. *Programming in Higher-Order Typed Lambda-Calculi.* Research Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, 1989.

[36] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[37] S-Å. Tärnlund. Horn clause computability. *BIT*, 17:215–226, 1977.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399